

AustinH ML2 FINAL Report

by Austin Hooper

FILE	936_AUSTIN_HOOPER_AUSTINH_ML2_FINAL_REPORT_316598_27895515.PDF (2.08M)		
TIME SUBMITTED	22-JAN-2021 04:41PM (UTC+0000)	WORD COUNT	6803
SUBMISSION ID	143111303	CHARACTER COUNT	41142



L7: Application of Machine Learning (ACCA7025)

Machine Learning Assignment 2

Student: Austin Hooper (1700481)

MSc Software Engineering and Artificial Intelligence

Wales Institute for Science and Art (WISA)
University of Wales Trinity Saint David

Lecturer: Gordon Dickers

18th January 2021

Table of Contents

1. Introduction	1
2. Key Developments in Neural Networks.....	2
3. Data Examination.....	3
4. Initial Model and Configuration	4
4.1. LeNet-5 Architecture.....	1
4.2. LeNet-5 Performance	3
5. Experiments.....	6
5.1. Initial Modifications	6
5.2. Imbalanced Data Handling	6
5.2.1. Resampling	6
5.2.2. Class Weights	7
5.3. Regularisation and Overfitting Experiments.....	8
5.4. Layer Experiments.....	10
5.5. Optimiser Experiments	11
6. Final Model	12
7. Conclusion and Future Work	17
References	18
Background Reading	22
Appendices.....	23
Appendix A	23
Appendix B	34

List of Figures

Figure 1 - LeNet-5 [29].....	1
Figure 2: LeNet-5 in Keras.....	2
Figure 3: Initial LeNet-5 accuracy per epoch	3
Figure 4: Initial LeNet-5 loss per epoch	4
Figure 5: Initial LeNet-5 confusion matrix.....	5
Figure 6: Initial LeNet-5 ROC curve.....	5
Figure 7: Class weights code.....	7
Figure 8: Dropout [38].....	9
Figure 9: Final model structure	12
Figure 10: Final model implementation.....	13
Figure 11: Final model accuracy per epoch.....	13
Figure 12: Final model loss per epoch.....	14
Figure 13: Final model confusion matrix.....	15
Figure 14: Final model ROC curve.....	16

List of Tables

Table 1: Dataset class summary.....	3
Table 2: Initial model comparison	4
Table 3: LeNet-5 layer architecture [1, p. 463].....	1
Table 4: Initial LeNet-5 test set performance	4
Table 5: Regularisation experiments.....	8
Table 6: Dropout experiments.....	9
Table 7: Activation function experiments	10
Table 8: Momentum and LR experiments	11
Table 9: Final model classification report.....	14

Abstract

This research reviewed and explored Convolutional Neural Networks for facial expression recognition on the FER-2013 dataset using Python. The dataset was examined and described, and a Neural Network was developed based on LeNet-5. Adjusting the number of hidden layers and neurons in each improved performance. Regularisation, primarily dropout layers, further improved performance. The initial loss function of categorical cross entropy was found to perform best. Resampling and class weights were explored, with no impact on performance. Evaluated on unseen test data, the final model achieved an accuracy of 60% and Area Under the Curve micro-average of 0.76. The identified future work consists of further examination of each class' data, data augmentation, and implementing a custom loss function.

1. Introduction

Artificial Neural Networks (ANNs) are a subset of Machine Learning (ML) which are simplified models of the brain [1, p. 445]. This assignment explores image classification using a Convolutional Neural Network (CNN) using the TensorFlow [2], Keras [3] and Scikit-Learn [4] Python libraries. Key developments in ANNs are reviewed in Section 2, together with related work. The dataset is of facial expressions and emotions, analysed in Section 3. LeNet-5 was chosen as a base model, benchmarked in Section 4. Experiments are detailed in Section 5. The final model is described and benchmarked in Section 6. Section 7 concludes the document. Code and dependencies are included in Appendix A.

2. Key Developments in Neural Networks

ANNs are based on the McCulloch-Pitts neuron, proposed in 1943 [5]. These neurons perform logical operations, though are unable to learn. The Rosenblatt Perceptron [6] is more sophisticated, featuring weighted connections, thresholds, inhibitory and excitatory inputs. Activation functions are applied to these inputs to produce an output.

Deep Neural Networks (DNN) consist of many hidden layers [1, pp. 27–29], they are simplified models of the cerebral cortex [1, p. xv]. The name “deep learning” was coined by Hinton *et al.* in their 2006 paper detailing training a model capable of accurately predicting the value of hand-written digits [7]. DNNs perform especially well with large amounts of training data, possible due to increased compute performance in recent hardware.

Convolutional Neural Networks (CNNs) were popularised by LeCun *et al.* when applied to handwritten digit recognition [8]. CNNs are often used for image recognition. Contrary to DNNs, CNNs are based on the visual cortex [1, p. 445]. CNNs consist of convolutional layers which learn filters from images, passed to pooling layers for dimensionality reduction, followed by fully connected layers and an output.

ANNs have a range of hyperparameters. In ANNs, Learning Rate (LR) controls the rate to which the model adapts to data, moving towards a lower loss value [9]. In CNNs, kernel size determines filter size when learning filters from images. Unit count controls the number of neurons used for learning, increasing model complexity [10]. Activation functions determine whether a neuron should be fired given input data – and thus must be chosen carefully. Regularisation discourages complexities, preventing overfitting.

Facial Expression Recognition (FER) has been extensively explored, especially recently, using DNNs and CNNs [11]–[17]. These papers highlight variance in classification accuracy across emotions. Research articles also highlight the difficulty of training on the dataset used in this assignment. State-of-the-art algorithms were found to achieve between 60-75% accuracy [18], [19]. Some researchers developed bespoke CNNs, achieving 50-60% accuracy, further highlighting the class imbalance and resulting accuracy variations [12], [20].

3. Data Examination

The FER-2013 dataset [21] consists of 35,887 images of human faces across seven emotions. It is split into test and train directories, each containing one directory per class. The dataset contains no validation directory, so this was managed using the ImageDataGenerator validation_split parameter. Details of the dataset's classes are shown in Table 1.

Table 1: Dataset class summary

Class Name	No. Train	No. Test	No. Total
Angry	3,995	958	4,953
Disgust	436	111	547
Fear	4,097	1,024	5,121
Happy	7,215	1,774	8,989
Neutral	4,965	1,233	6,198
Sad	4,830	1,247	6,077
Surprise	3,171	831	4,002

The number of images in each class varies dramatically - from 547 to 8,989. The train:test split is consistent across classes, at approximately 4:1 / 80:20. All classes other than Disgust were used for classification, this class had very few images. Remaining class sizes ranged from 4,953 to 8,989 images, potentially resulting in those with more images being overfitted.

The images are JPG format, approximately 1-2KB, 8-bit greyscale, 48×48 pixels resolution. Low resolution and lack of colour decreases training efficacy, though also considerably decreases training time. Investigation showed that ImageDataGenerator's pixel values ranged from 0-1 as opposed to 0-255, thus normalisation was not necessary.

Some images are incorrectly labelled, or invalid – for example, pure black images and error symbols. This has drawbacks, however, it could aid in generalisation and reduce overfitting. Each class' images were not balanced regarding ethnicity, age, or gender. This varies by class, for example, the Happy class appears to be more balanced. This could impact real-world efficacy of the system as it is not a true representation of people across the world.

4. Initial Model and Configuration

The default LR of 0.01 was used with the Stochastic Gradient Descent (SGD) optimiser. Early stopping was used with a patience of 5 epochs, to stop training once validation loss stops improving. Another callback was used to reduce LR on plateaus, with a factor of 0.3 and patience of 2 epochs. Images were not resized as training was fast at the default 48x48 resolution.

Default evaluation metrics of loss and accuracy were used. Research showed that Area Under the Curve (AUC) scores are considered more appropriate for imbalanced data [22]–[25]. However, this dataset is multi-class, thus Keras' AUC metric, which uses continuous class probabilities, significantly overestimates model performance.

This assignment required a starting point architecture to be chosen. LeNet-5 was chosen due to its initial performance relative to other models explored. Its relative simplicity is beneficial due to the low number of images in training, likely to reduce overfitting. Three other models were tested: ResNet-34 [26], AlexNet [27], and VGG-16 [28]. These are detailed in Table 2.

Table 2: Initial model comparison

Model Name	Authors	Year Proposed	Model Structure (adapted to fit dataset)	Initial Model Accuracy (%)	Altered* Model Accuracy (%)
ResNet-34	K. He, X. Zhang, S. Ren, J. Sun	2016	1 × Conv2D: 64, 3×3 1 × MaxPooling2D: 2×2 6 × Conv2D: 64, 3×3 8 × Conv2D: 128, 3×3 12 × Conv2D: 256, 3×3 6 × Conv2D: 512, 3×3 1 × AvgPooling2D: 2×2 1 × Dense: 1000 1 × Dense Softmax: 6	28	30
AlexNet	A. Krizhevsky, I. Sutskever, G. E. Hinton	2012	1 × Conv2D: 96, 3×3 1 × Conv2D: 256, 3×3 1 × MaxPooling2D: 2×2 1 × Conv2D: 384, 3×3 1 × MaxPooling2D: 2×2 1 × Conv2D: 384, 3×3 1 × Conv2D: 256, 3×3 1 × MaxPooling2D: 2×2 1 × Dense: 4096 1 × Dense: 4096 1 × Dense: 1000 1 × Dense Softmax: 6	35	45
VGG-16	K. Simonyan, A. Zisserman	2015	2 × Conv2D: 64, 3×3 1 × MaxPooling2D: 2×2 2 × Conv2D: 128, 3×3 1 × MaxPooling2D: 2×2 3 × Conv2D: 256, 3×3 1 × MaxPooling2D: 2×2 3 × Conv2D: 512, 3×3 1 × MaxPooling2D: 2×2 3 × Conv2D: 512, 3×3 1 × MaxPooling2D: 2×2 1 × Dense: 4096 1 × Dense: 4096 1 × Dense Softmax: 6	36	44

* Altered models were modified to better suit the scenario. Modifications consisted of adding pooling and dropout, along with modifying the number of hidden layers and their number of units.

4.1. LeNet-5 Architecture

LeNet-5, developed by LeCun *et al.* [29], consists of alternating convolution and subsampling layers, followed by fully connected layers, and an output layer, shown in Figure 1. The architecture's layers are detailed by Géron [1, p. 463], shown in Table 3. Its Keras implementation is shown in Figure 2.

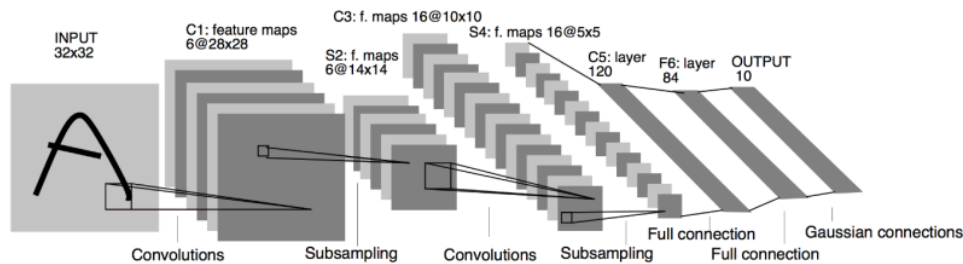


Figure 1 - LeNet-5 [29]

Table 3: LeNet-5 layer architecture [1, p. 463]

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully connected	–	10	–	–	RBF
F6	Fully connected	–	84	–	–	tanh
C5	Convolution	120	1×1	5×5	1	tanh
S4	Avg pooling	16	5×5	2×2	2	tanh
C3	Convolution	16	10×10	5×5	1	tanh
S2	Avg pooling	6	14×14	2×2	2	tanh
C1	Convolution	6	28×28	5×5	1	tanh
In	Input	1	32×32	–	–	–

```

# BASE LeNet-5
model = Sequential([
    # Convolutions
    Conv2D(filters=6, kernel_size=(5, 5), activation="tanh", input_shape=(48,48,1), strides=1),
    AveragePooling2D(pool_size=(2, 2), strides=1),

    Conv2D(filters=16, kernel_size=(5, 5), activation="tanh", strides=1),
    AveragePooling2D(pool_size=(2, 2), strides=1),

    Conv2D(filters=120, kernel_size=(5, 5), activation="tanh", strides=1),

    # Flatten to 1 dimensional
    Flatten(),

    # Fully connected layers
    Dense(units=84, activation="tanh"),
    Dense(units=6, activation="softmax"),
])

```

Figure 2: LeNet-5 in Keras

The default implementation features three convolutional layers, learning 6, 16 and 120 filters, respectively. They all feature kernel sizes of 5×5. Average pooling layers follow each convolution, which downsample the image. After flattening into 1D data, data is passed to two fully connected dense layers. The first features 84 units, and like all previous layers, uses the hyperbolic tangent activation function. Finally, a dense layer with 6 units (one per class) is used, with softmax activation, facilitating the output of class predictions.

4.2. LeNet-5 Performance

The model reached only ~37% validation accuracy, as shown in Figure 3. Despite LR modifications, improvement was negligible, thus training was limited to 50 epochs. This plateau is mirrored in loss values shown in Figure 4.

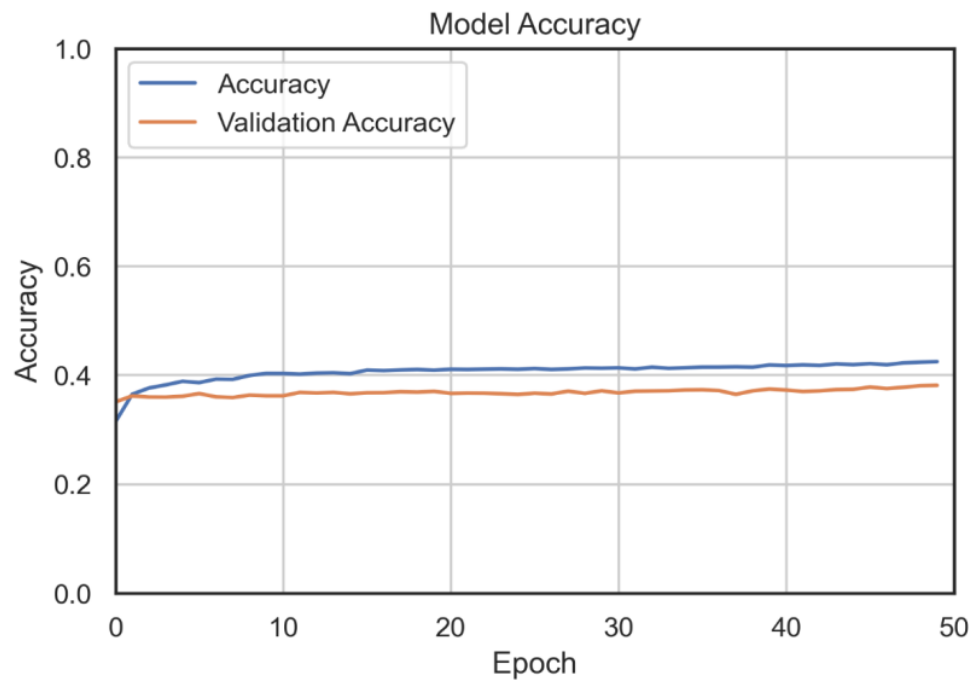


Figure 3: Initial LeNet-5 accuracy per epoch

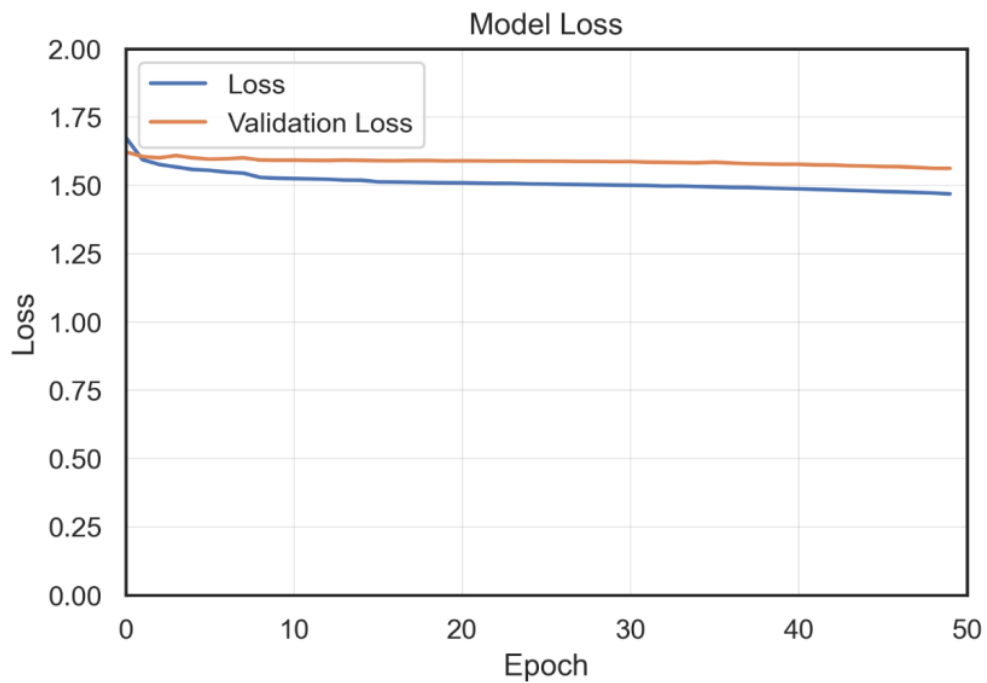


Figure 4: Initial LeNet-5 loss per epoch

The trained model was evaluated using unseen test data. The classification report is shown in Table 4.

Table 4: Initial LeNet-5 test set performance

	precision	recall	f1-score	support
angry	0.28	0.16	0.20	958
fear	0.27	0.13	0.17	1024
happy	0.47	0.69	0.55	1774
neutral	0.39	0.34	0.36	1233
sad	0.30	0.37	0.33	1247
surprise	0.52	0.52	0.52	831
accuracy			0.40	7067
macro avg	0.37	0.37	0.36	7067
weighted avg	0.38	0.40	0.37	7067

Test results were also evaluated using a confusion matrix (Figure 5) and Receiver Operating Characteristic (ROC) curve (Figure 6). Confusion matrix and AUC scores highlight a significant variance in classification performance per class, potentially due to the varying number of images per class. The angry and fear classes were classified particularly poorly. The micro-average AUC score was 0.64.



Figure 5: Initial LeNet-5 confusion matrix

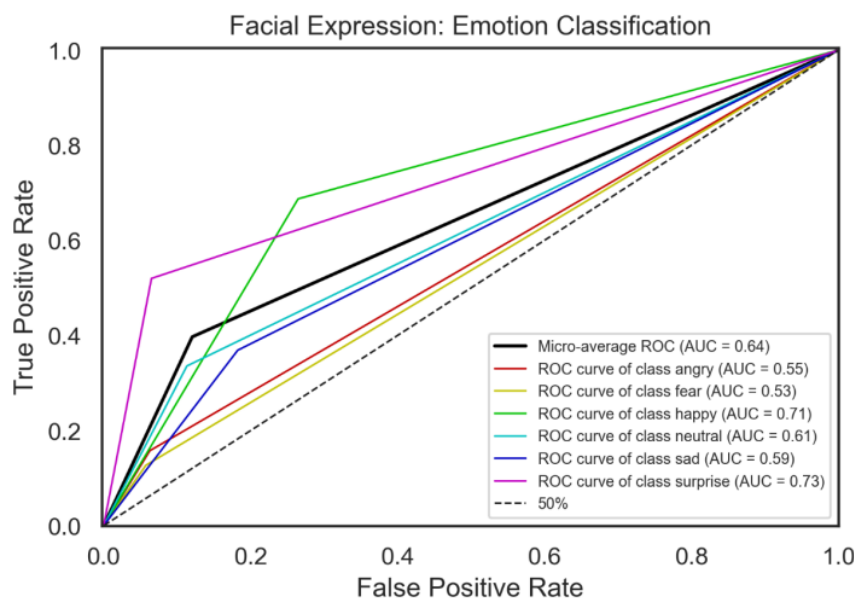


Figure 6: Initial LeNet-5 ROC curve

5. Experiments

A range of experiments were conducted, detailed in this section. Given the random nature of training, stated accuracy values are averages of three runs. All stated validation accuracy improvements were also reflected on the test set.

5.1. Initial Modifications

The default hyperbolic tangent activation function prevented progress of the model – changing other parameters had no effect. This was replaced with the Rectified Linear Unit (ReLU) – improving validation accuracy by ~8% to ~45%.

Kernel size was reduced from 5x5 to 3x3 due to the low resolution of the input images. This increased validation accuracy by ~1% to ~46%, however, this could potentially improve further training.

Substituting AveragePooling2D for MaxPooling2D increased validation accuracy by ~5% to ~51%, probably due to the high contrast of the images – max pooling uses the highest pixel value, as opposed to mean [30], [31].

5.2. Imbalanced Data Handling

5.2.1. Resampling

Resampling involves modifying class distribution of a dataset [32]. Facilitated by the Imbalanced-Learn library [33], the process is detailed in Appendix B.

Random over sampling, under sampling, and Synthetic Minority Oversampling Technique (SMOTE) [34] were tested. Oversampling involves randomly duplicating items from the minority classes so that their sizes match the majority class, while under sampling randomly deletes until all classes are the size of the minority class [32], [35]. SMOTE creates new samples from existing ones. These methods did not benefit loss or accuracy.

5.2.2. Class Weights

Class weights are used to weight minority classes heavier than majority classes, counteracting class imbalance [36]. See in Figure 7 code for retrieving class weights, and its result.

```
# Use sklearn to get class weights as a list
class_weights_list = class_weight.compute_class_weight(
    'balanced',
    np.unique(training_set.classes),
    training_set.classes)

# Convert into a dictionary in the format of class1:weight1, class2:weight2...
class_weights_dict = {}

# For each class, add it and its value to the dictionary
for class_num in range(0, num_classes):
    class_weights_dict[class_num] = class_weights_list[class_num]

{
0: 1.1795473508552357,
1: 1.1500406752084604,
2: 0.6531242781242781,
3: 0.9756297446514838,
4: 1.4859414006043883,
5: 0.9491020476670023
}
```

Figure 7: Class weights code

When these weights were passed to the model fit function, performance remained the same at ~51%. Both resampling and class weights were ineffective. The reason for this is unclear, though potentially due to the quality of images and their labelling. For example, many Disgust images appeared to fit better to the Sad class.

5.3. Regularisation and Overfitting Experiments

No layer modifications increased validation accuracy, despite accuracy rising to 80%. As such, regularisation experiments were conducted first.

Adding noise using Keras' GaussianNoise layer had no impact, as did data augmentation on the train set.

In Keras, the primary forms of regularisation are dropout layers, and regulariser hyperparameters, as listed by the Keras documentation [37]:

- kernel_regularizer
- bias_regularizer
- activity_regularizer

Only activity_regularizer=l2(0.0001) was beneficial, improving validation accuracy by ~2% to ~53%, shown in Table 5.

Table 5: Regularisation experiments

activity_regularizer experiments with l2(0.0001)	
activity_regularizer Placement	Val. Accuracy (%, 1DP)
None	51.0
On Conv2D layers except input	52.1
On Conv2D layers including input	52.5
On all Conv2D and Dense except output	53.0
On all Conv2D and Dense including output	53.2

Another overfitting countermeasure tested was dropout layers. Described by Hinton *et al.* in 2014 [38] and later patented by Google [39], dropout involves randomly dropping units and their connections during training, shown in Figure 8.

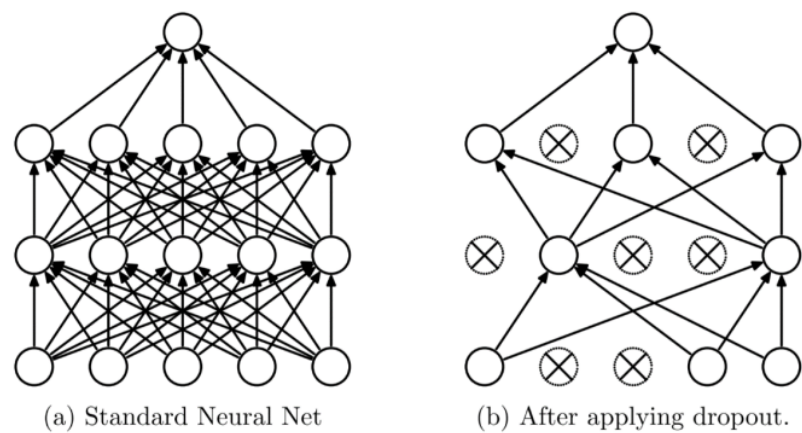


Figure 8: Dropout [38]

Dropout was most effective when used after every convolutional and dense layer; 0.15 was most beneficial, shown in Table 6.

Table 6: Dropout experiments

Dropout experiments	
Dropout after Conv2D and Dense	Val. Accuracy (%, 1DP)
None	53.2
Value: 0.10	54.5
Value: 0.15	55.7
Value: 0.20	52.7
Value: 0.25	50.7

5.4. Layer Experiments

Additional layers with the same filter count did not improve performance. The model reached the same final validation accuracy, though taking more epochs to do so.

Filter and unit count experiments provided small improvements. Replacing convolution filter counts of 6, 16, and 120 with 32, 128, and 256 improved validation accuracy by ~3% to ~58%. Filter counts above 256 had no impact. Modifying the number of units in the Dense layer provided no improvement.

After altering layer filter counts, the number of layers was revisited. Additional dense layers slowed the model with no accuracy improvement. Adding a duplicate layer after the 128-filter convolution improved performance by ~1% to ~59%. Dropout was altered, having modified layers – with no improvement.

No tested activation function improved the model, as shown in Table 7.

Table 7: Activation function experiments

Activation Function	Val. Accuracy (%)	Comment
BASE: ReLU	59%	Initial activation
SELU	54%	Learned far faster but plateaued
ELU	54%	Learned far faster but plateaued
Leaky ReLU	56-58%	Same value at alpha = 0.1, 0.2, 0.3 (default) and 0.5
PReLU	58%	Learned at similar rate to ReLU
Softplus	26%	Did not learn beyond first epoch accuracy
Sigmoid	31%	Learned very slowly
Softsign	54%	Learned at similar rate to ReLU
Exponential	14%	Did not learn

No weight initialiser from [40] improved performance, thus, default kernel_initializer and bias_initializer values of "glorot_uniform" and "zeros" were used, respectively.

5.5. Optimiser Experiments

Learning rate and momentum adjustments produced no improvement, shown in Table 8. Momentum is most effective for escaping local optimums at a high value with low LR [1, pp. 352–353].

Table 8: Momentum and LR experiments

Learning Rate	Momentum	Val. Accuracy (%)	Comment
BASE: 0.01	BASE: N/A	59%	~120 epochs.
0.1	N/A	59%	Reached after only 25-30 epochs. Accuracy rose to 80%. Increasing regularisation had no impact on val. accuracy.
0.15	N/A	59%	
0.05	N/A	59%	
0.01	0.99	25%	Did not learn.
	0.9	57%	~120 epochs as with base.
	0.7	57%	~120 epochs as with base.
0.001	0.99	25%	Did not learn.
	0.9	59%	~120 epochs as with base.
	0.7	57%	Very slow, ~250 epochs.
0.0001	0.99	57%	~120 epochs as with base.
	0.9	54%	Very slow, ~450 epochs.
	0.7	45%	Very slow, plateaued at ~550 epochs

None of Keras' KLDivergence, Poisson, CategoricalHinge or CosineSimilarity loss functions improved performance over categorical cross entropy. Furthermore, no other Keras optimiser improved performance over SGD.

6. Final Model

The model's structure is shown below in Figure 9, along with its Keras implementation in Figure 10.

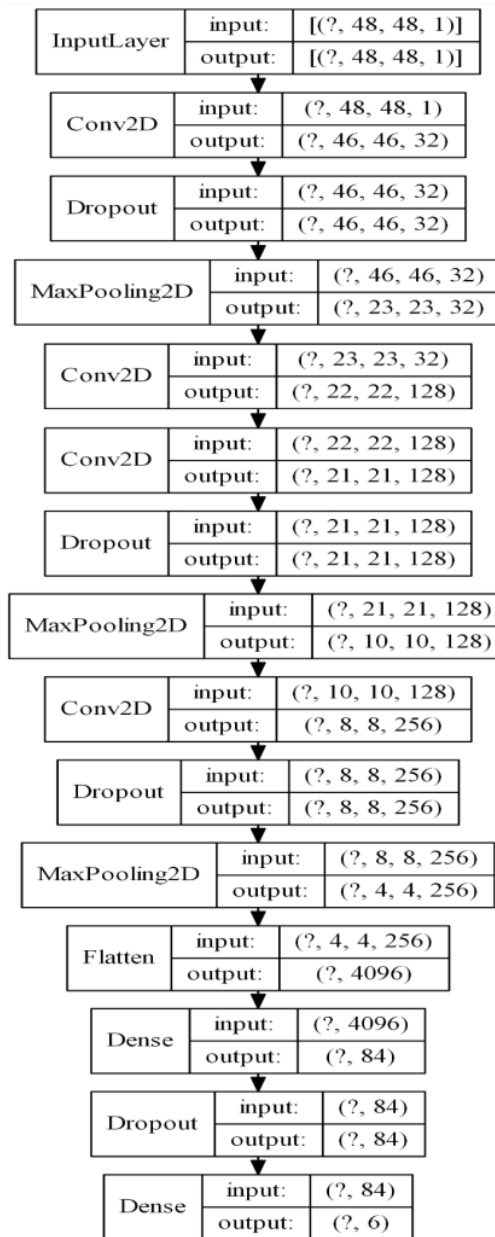


Figure 9: Final model structure

```

model = Sequential([
    # Convolutions and pooling
    Conv2D(32, (3,3), activation='relu', input_shape=(48,48,1), activity_regularizer=l2(0.0001)),
    Dropout(0.15),
    MaxPooling2D((2, 2)),

    Conv2D(128, (2, 2), activation='relu', activity_regularizer=l2(0.0001)),
    Conv2D(128, (2, 2), activation='relu', activity_regularizer=l2(0.0001)),
    Dropout(0.15),
    MaxPooling2D((2, 2)),

    Conv2D(256, (3,3), activation='relu', activity_regularizer=l2(0.0001)),
    Dropout(0.15),
    MaxPooling2D((2, 2)),

    # Flatten to 1D data
    Flatten(),

    # Dense
    Dense(84, activation='relu', activity_regularizer=l2(0.0001)),
    Dropout(0.15),

    # Output
    Dense(num_classes, activation='softmax', activity_regularizer=l2(0.0001))
])

```

Figure 10: Final model implementation

The model achieved an accuracy of 60% on unseen data. Training accuracy and loss are shown in Figures 11 and 12.

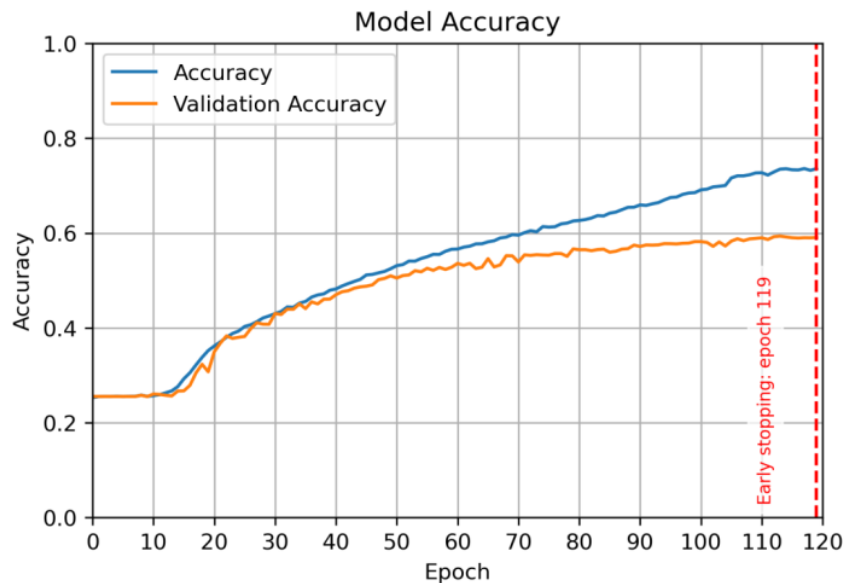


Figure 11: Final model accuracy per epoch

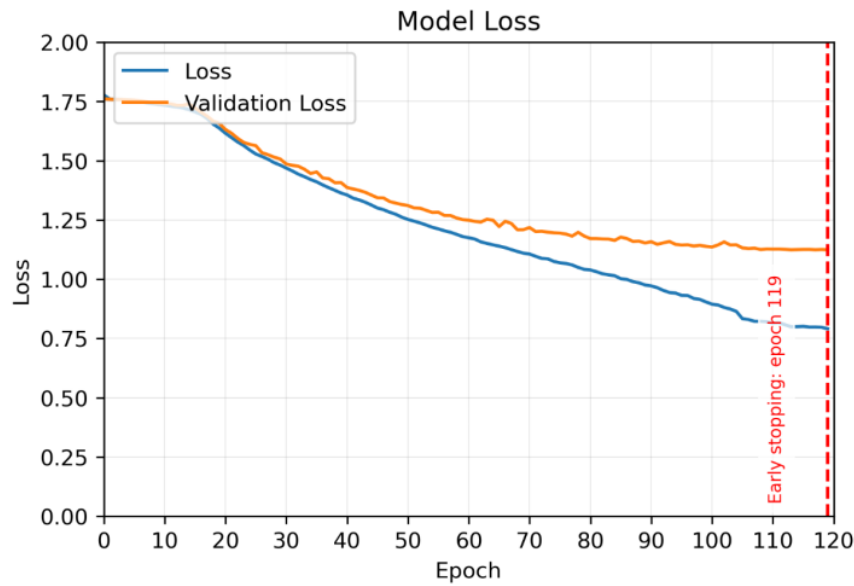


Figure 12: Final model loss per epoch

Shown in Table 9, precision, recall and F1 scores significantly improved over the base model. The confusion matrix in Figure 13 highlights the improved performance across all classes, especially Angry and Fear.

Table 9: Final model classification report

	precision	recall	f1-score	support
angry	0.53	0.49	0.51	958
fear	0.46	0.29	0.36	1024
happy	0.78	0.82	0.80	1774
neutral	0.51	0.63	0.56	1233
sad	0.46	0.49	0.47	1247
surprise	0.75	0.72	0.73	831
accuracy			0.60	7067
macro avg	0.58	0.57	0.57	7067
weighted avg	0.59	0.60	0.59	7067

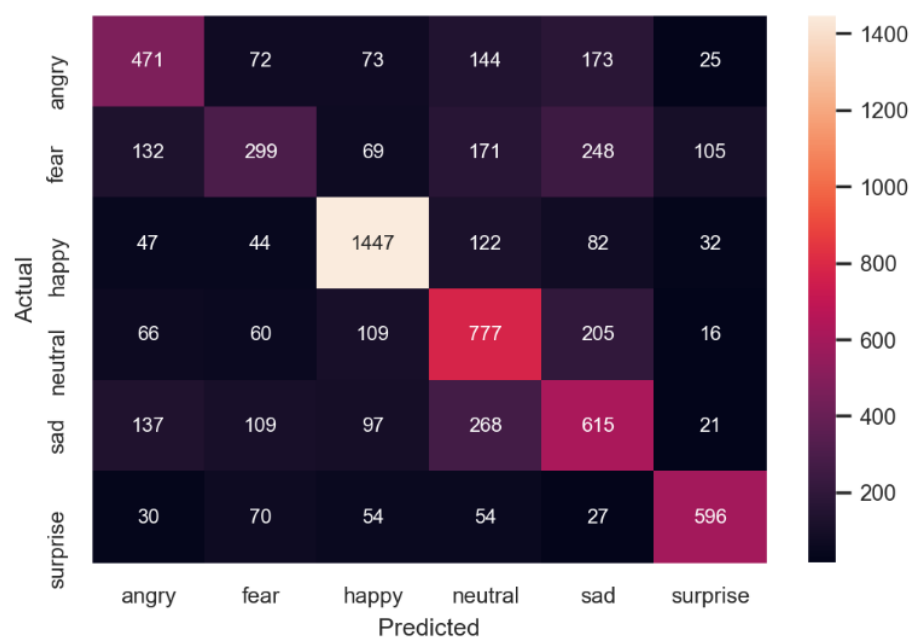


Figure 13: Final model confusion matrix

Shown in Figure 14, the model achieved a micro-average AUC score of 0.76 compared to the initial 0.64.

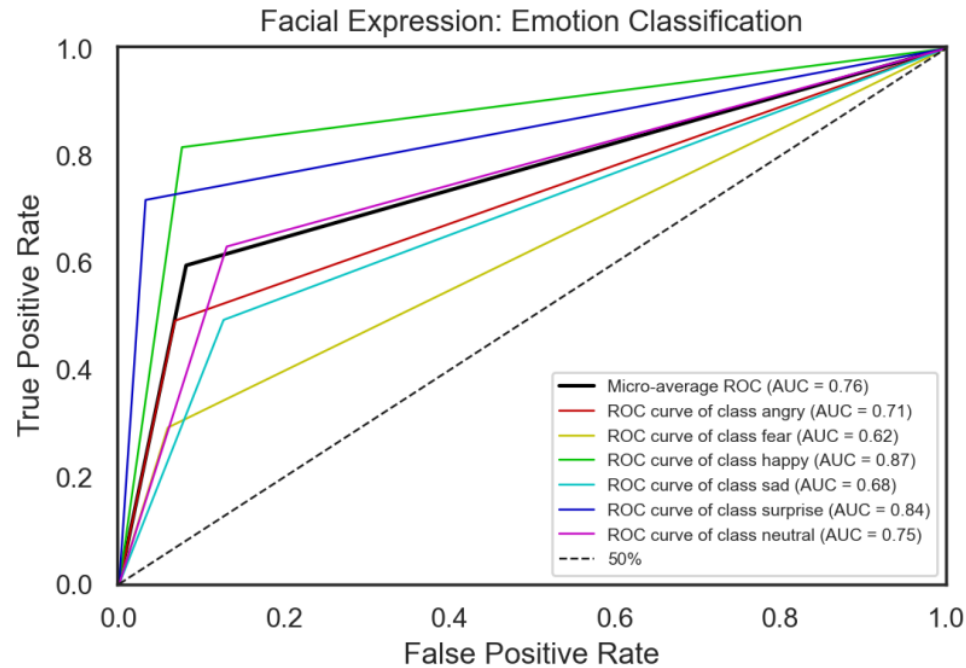


Figure 14: Final model ROC curve

As Angry and Sad were classified worse than others, the model was tested without them, achieving 72% accuracy on the 4 remaining classes.

7. Conclusion and Future Work

A CNN was developed for classifying emotions from facial expressions, based on LeNet-5. The model was successful, achieving an accuracy of 60% across six classes, despite the challenging dataset. A wide range of ML techniques has been adopted, many having no significant impact on performance.

Future work could incorporate further examination of the dataset, to aid in developing a better model. The developed model could be validated using additional emotion datasets. The model could be adapted and tested for a broader range of applications. Further ML methods could be explored such as custom loss functions, additional layer hyperparameters, and data augmentation.

References

- [1] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly, 2019.
- [2] TensorFlow, "TensorFlow." <https://www.tensorflow.org/> (accessed Dec. 27, 2020).
- [3] Keras, "Keras: the Python deep learning API." <https://keras.io/> (accessed Dec. 27, 2020).
- [4] scikit-learn, "scikit-learn: machine learning in Python — scikit-learn 0.23.2 documentation." <https://scikit-learn.org/stable/> (accessed Nov. 10, 2020).
- [5] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bull. Math. Biophys.*, vol. 5, no. 4, pp. 115–133, 1943, doi: 10.1007/BF02478259.
- [6] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychol. Rev.*, vol. 65, no. 6, pp. 386–408, 1958, doi: 10.1037/h0042519.
- [7] G. Hinton, S. Osindero, and Y.-W. Teh, "A Fast Learning Algorithm for Deep Belief Nets," *Neural Comput.*, vol. 18, pp. 1527–1554, 2006, doi: 10.1162/neco.2006.18.7.1527.
- [8] Y. LeCun *et al.*, "Handwritten digit recognition: applications of neural network chips and automatic learning," *IEEE Commun. Mag.*, vol. 27, no. 11, pp. 41–46, Nov. 1989, doi: 10.1109/35.41400.
- [9] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A Search Space Odyssey," *IEEE Trans. Neural Networks Learn. Syst.*, vol. 28, no. 10, pp. 2222–2232, Oct. 2017, doi: 10.1109/TNNLS.2016.2582924.
- [10] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proc. Natl. Acad. Sci.*, vol. 79, no. 8, pp. 2554–2558, 1982, doi: 10.1073/pnas.79.8.2554.
- [11] Lin, Wang, and Wu, "Multiple Convolutional Neural Networks Fusion Using Improved Fuzzy Integral for Facial Emotion Recognition," *Appl. Sci.*, vol. 9, no. 13, p. 2593, 2019.
- [12] W. H. Abdulsalam, R. Al-hamdani, and M. Abdullah, "Facial Emotion Recognition from Videos Using Deep Convolutional Neural Networks," *Int. J. Mach. Learn. Comput.*, vol. 9, pp. 14–19, 2019.

- [13] A. Mollahosseini, D. Chan, and M. H. Mahoor, "Going deeper in facial expression recognition using deep neural networks," in *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*, Mar. 2016, pp. 1–10, doi: 10.1109/WACV.2016.7477450.
- [14] F. Zhang, T. Zhang, Q. Mao, and C. Xu, "Joint Pose and Expression Modeling for Facial Expression Recognition," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Jun. 2018, pp. 3359–3368, doi: 10.1109/CVPR.2018.00354.
- [15] X. Chen, D. Li, P. Wang, and X. Yang, "A Deep Convolutional Neural Network With Fuzzy Rough Sets for FER," *IEEE Access*, vol. 8, pp. 2772–2779, 2020, doi: 10.1109/ACCESS.2019.2960769.
- [16] G. Verma and H. Verma, "Hybrid-Deep Learning Model for Emotion Recognition Using Facial Expressions," *Rev. Socionetwork Strateg.*, vol. 14, no. 2, pp. 171–180, 2020, doi: 10.1007/s12626-020-00061-6.
- [17] S. Xie, H. Hu, and Y. Wu, "Deep multi-path convolutional neural network joint with salient region attention for facial expression recognition," *Pattern Recognit.*, vol. 92, pp. 177–191, 2019, doi: <https://doi.org/10.1016/j.patcog.2019.03.019>.
- [18] S. Li and W. Deng, "Deep Facial Expression Recognition: A Survey," *IEEE Trans. Affect. Comput.*, p. 1, 2020, doi: 10.1109/TAFFC.2020.2981446.
- [19] A. Agrawal and N. Mittal, "Using CNN for facial expression recognition: a study of the effects of kernel size and number of filters on accuracy," *Vis. Comput.*, vol. 36, 2020, doi: 10.1007/s00371-019-01630-9.
- [20] W. Xie, X. Jia, L. Shen, and M. Yang, "Sparse deep feature learning for facial expression recognition," *Pattern Recognit.*, vol. 96, p. 106966, 2019, doi: <https://doi.org/10.1016/j.patcog.2019.106966>.
- [21] Wolfram Research, "Facial Expression Recognition 2013 (FER-2013) Dataset," 2018. <https://datarepository.wolframcloud.com/resources/FER-2013> (accessed Jan. 11, 2021).
- [22] Y. Tang, Y. Zhang, N. V Chawla, and S. Krasser, "SVMs Modeling for Highly Imbalanced Classification," *IEEE Trans. Syst. Man, Cybern. Part B*, vol. 39, no. 1, pp. 281–288, Feb. 2009, doi: 10.1109/TSMCB.2008.2002909.
- [23] Q. Zou, S. Xie, Z. Lin, M. Wu, and Y. Ju, "Finding the Best Classification Threshold in Imbalanced Classification," *Big Data Res.*, vol. 5, pp. 2–8, 2016, doi: <https://doi.org/10.1016/j.bdr.2015.12.001>.

- [24] Q. Kang, X. Chen, S. Li, and M. Zhou, "A Noise-Filtered Under-Sampling Scheme for Imbalanced Classification," *IEEE Trans. Cybern.*, vol. 47, no. 12, pp. 4263–4274, Dec. 2017, doi: 10.1109/TCYB.2016.2606104.
- [25] S. S. Mullick, S. Datta, S. G. Dhekane, and S. Das, "Appropriateness of performance indices for imbalanced data classification: An analysis," *Pattern Recognit.*, vol. 102, p. 107197, 2020, doi: <https://doi.org/10.1016/j.patcog.2020.107197>.
- [26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 770–778, doi: 10.1109/CVPR.2016.90.
- [27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, 2012, vol. 25, pp. 1097–1105, [Online]. Available: <https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>.
- [28] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *3rd Int. Conf. Learn. Represent. ICLR 2015 - Conf. Track Proc.*, pp. 1–14, 2015.
- [29] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998, doi: 10.1109/5.726791.
- [30] P. Zhou, Z. Qi, S. Zheng, J. Xu, H. Bao, and B. Xu, "Text Classification Improved by Integrating Bidirectional {LSTM} with Two-dimensional Max Pooling," in *Proceedings of {COLING} 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, Dec. 2016, pp. 3485–3495, [Online]. Available: <https://www.aclweb.org/anthology/C16-1329>.
- [31] S. R. Bulò, G. Neuhold, and P. Kotschieder, "Loss Max-Pooling for Semantic Image Segmentation," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017, pp. 7082–7091, doi: 10.1109/CVPR.2017.749.
- [32] S. Pouyanfar *et al.*, "Dynamic Sampling in Convolutional Neural Networks for Imbalanced Data Classification," in *2018 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, Apr. 2018, pp. 112–117, doi: 10.1109/MIPR.2018.00027.
- [33] G. Lemaitre, F. Nogueira, D. Oliveira, and C. Aridas, "imbalanced-learn API — imbalanced-learn 0.5.0 documentation," 2017. <https://imbalanced-learn.readthedocs.io/en/stable/api.html> (accessed Jan. 01, 2021).

- [34] N. V Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority over-Sampling Technique," *J. Artif. Int. Res.*, vol. 16, no. 1, pp. 321–357, Jun. 2002.
- [35] M. S. Reza and J. Ma, "Imbalanced Histopathological Breast Cancer Image Classification with Convolutional Neural Network," in *2018 14th IEEE International Conference on Signal Processing (ICSP)*, Aug. 2018, pp. 619–624, doi: 10.1109/ICSP.2018.8652304.
- [36] M. Zhu *et al.*, "Class Weights Random Forest Algorithm for Processing Class Imbalanced Medical Data," *IEEE Access*, vol. 6, pp. 4641–4652, 2018, doi: 10.1109/ACCESS.2018.2789428.
- [37] Keras, "Layer weight regularizers." <https://keras.io/api/layers/regularizers/> (accessed Jan. 06, 2021).
- [38] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014.
- [39] Geoffrey E. Hinton, Alexander Krizhevsky, Ilya Sutskever, and Nitish Srivastva, "SYSTEM AND METHOD FOR ADDRESSING OVERFITTING IN A NEURAL NETWORK," US 9.406,017 B2, Aug. 02, 2016.
- [40] Keras, "Layer weight initializers." <https://keras.io/api/layers/initializers/> (accessed Jan. 19, 2021).

Background Reading

A. Ali-Gombe and E. Elyan, "MFC-GAN: Class-imbalanced dataset classification using Multiple Fake Class Generative Adversarial Network," *Neurocomputing*, vol. 361, pp. 212–221, 2019, doi: <https://doi.org/10.1016/j.neucom.2019.06.043>.

B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Comput. Math. Math. Phys.*, vol. 4, no. 5, pp. 1–17, 1964, doi: [https://doi.org/10.1016/0041-5553\(64\)90137-5](https://doi.org/10.1016/0041-5553(64)90137-5).

B. Yousuf, A. Shukla, M. K. Arora, A. Bindal, and A. S. Jasrotia, "On Drivers of Subpixel Classification Accuracy—An Example From Glacier Facies," *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.*, vol. 13, pp. 601–608, 2020, doi: [10.1109/JSTARS.2019.2955955](https://doi.org/10.1109/JSTARS.2019.2955955).

J.-W. Huang, C.-W. Chiang, and J.-W. Chang, "Email security level classification of imbalanced data using artificial neural network: The real case in a world-leading enterprise," *Eng. Appl. Artif. Intell.*, vol. 75, pp. 11–21, 2018, doi: <https://doi.org/10.1016/j.engappai.2018.07.010>.

S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Essex: Pearson Education, 2016.

V. Podlozhnyuk, "Image Convolution with CUDA," 2007. [Online]. Available: https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf.

W. Byun, J.-M. Lee, Y. Kim, and T. A. Brusseau, "Classification Accuracy of a Wearable Activity Tracker for Assessing Sedentary Behavior and Physical Activity in 3–5-Year-Old Children," *Int. J. Environ. Res. Public Health*, vol. 15, no. 4, 2018, doi: [10.3390/ijerph15040594](https://doi.org/10.3390/ijerph15040594).

Appendices

Appendix A

Library dependencies:

- tensorflow
- keras
- pandas
- seaborn
- numpy
- sklearn
- matplotlib
- zipfile
- os
- time
- shutil
- imblearn
- PIL

Code included on the following pages.

Initial setup

```
import tensorflow as tf
import pandas as pd
import seaborn as sn
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow
from tensorflow import keras
from zipfile import ZipFile
import os
import time
import shutil
from keras.utils.vis_utils import plot_model
from numpy.random import seed
from sklearn.utils import class_weight
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc
from tensorflow.keras.models import Sequential
from tensorflow.keras.preprocessing.image import ImageDataGenerator, load_img
from tensorflow.keras.layers import Conv2D, Conv1D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization, Activation, AveragePooling2D
from tensorflow.keras.layers import SeparableConv2D, LeakyReLU, PReLU, Input, ZeroPadding2D
from tensorflow.keras.layers import GlobalMaxPooling2D, GaussianNoise
import tensorflow.keras.metrics
from keras.metrics import AUC
from keras.regularizers import l2, l1
from keras.callbacks import ReduceLROnPlateau
from tensorflow.keras import regularizers
from tensorflow.keras.optimizers import Adam, RMSprop, SGD, Adadelta
from imblearn.over_sampling import RandomOverSampler, SMOTE, ADASYN
from imblearn.under_sampling import RandomUnderSampler
from PIL import Image

print(f"TF ver: {tf.__version__}")
print(f"Keras ver: {keras.__version__}")
print("Num GPUs Available: ", len(tf.config.experimental.list_physical_devices('GPU'))))

%%html
<!-- Makes the container wider so the training log is easier to read -->
<style>
.container {
    width: 1400px !important;
}
</style>
```

Extract the dataset and load it using ImageDataGenerator

Use the validation_split parameter to split the train directory's contents into train and validate sets

```
# with ZipFile('dataset.zip', 'r') as zipObj:
#     # Extract all the contents of zip file in current directory
#     zipObj.extractall()

train_dir = './dataset/train/'
test_dir = './dataset/test/'

batch_size = 64

# Classify these classes
# "angry", "disgust", "fear", "happy", "sad", "surprise", "neutral"
classes_to_use = ["angry", "fear", "happy", "sad", "surprise", "neutral"]
num_classes = len(classes_to_use)

print("Train: ")
# Train set
train_datagen = ImageDataGenerator(rescale=1./255,
                                   validation_split = 0.2)

training_set = train_datagen.flow_from_directory(train_dir,
                                                batch_size=batch_size,
                                                target_size=(48,48),
                                                shuffle=True,
                                                color_mode='grayscale',
                                                class_mode='categorical',
                                                subset='training',
                                                classes=classes_to_use,
                                                seed=1)

print("\nValidation:")
# Validation set
validation_set = train_datagen.flow_from_directory(train_dir,
                                                  batch_size=batch_size,
                                                  target_size=(48,48),
                                                  shuffle=True,
                                                  color_mode='grayscale',
                                                  class_mode='categorical',
                                                  subset='validation',
                                                  classes=classes_to_use,
                                                  seed=1)

print("\nTest:")
# Test set
test_datagen = ImageDataGenerator(rescale=1./255)
test_set = test_datagen.flow_from_directory(test_dir,
                                           batch_size=batch_size,
                                           shuffle=True,
                                           target_size=(48,48),
                                           color_mode='grayscale',
                                           class_mode='categorical',
                                           classes=classes_to_use,
                                           seed=1)

print("\nClasses: ", test_set.class_indices)
```

Resampling

1. Firstly, get the data from ImageDataGenerator and split it into image data and classe labels, in a format that RandomOverSampler, RandomUnderSampler, and SMOTE can work with
2. Next, specify a sampling method and apply it to the data. Then we print out the number of images per class before and after sampling to check it's worked
3. Then, get each sampled 2D array back into image form and save it to a new dataset
4. Finally, read in the new, resampled train data using ImageDataGenerator. We don't need to re-load the test or validation set because that's unchanged

```
# # Step 1

# train_images = []
# train_classes = []

# # Get the data as lists for data and class labels
# for i in range(training_set.__len__()):
#     train_classes.append(
#         np.argmax(training_set.__getitem__(i)[1], axis=1) # [1] is the class label
#     )
#     train_images.append(
#         training_set.__getitem__(i)[0].reshape(48, 48).flatten() # [0] is the image data
#     )

# # Get as numpy arrays
# train_images = np.array(train_images)
# train_classes = np.array(train_classes).flatten()

# # Step 2

# # resampler = SMOTE(sampling_strategy='not majority') # Resample all except the one with the
# # most data
# # resampler = ADASYN(sampling_strategy='not majority', n_neighbors=5, n_jobs=-1) # Resample all
# # except the one with the most data
# # resampler = RandomOverSampler(sampling_strategy='not majority') # Resample all except the o
# # ne with the most data
# # resampler = RandomUnderSampler(sampling_strategy='not minority') # Resample all except the
# # one with the least data

# X_resampled, y_resampled = resampler.fit_resample(train_images, train_classes)

# # Check that the sampling has worked
# train_classes = train_classes.tolist()
# print("Number of images per class before sampling:")
# for i in range(0, 6):
#     print(train_classes.count(i))

# print("\nNumber of images per class after sampling:")
```

```

# for i in range (0, 6):
#     print(y_resampled.tolist().count(i))

# # Step 3

# # If exists from last time, delete to start over
# if os.path.exists(os.path.join(os.getcwd(), 'resampled_dataset')):
#     shutil.rmtree("resampled_dataset")

# os.mkdir("resampled_dataset")
# os.mkdir("resampled_dataset/train")

# # Create directory for each class
# for i in range (0, 6):
#     os.mkdir("resampled_dataset/train/" + str(i))

# # Save each image into their appropriate folder based on their y value
# for i in range(0, len(y_resampled)):
#     if i % 1000 == 0:
#         percent = round((i / len(y_resampled)) * 100)
#         print(str(percent) + "%")

#     tempImg = X_resampled[i]

#     # Convert pixel values from 0-1 to 0-255
#     for pixelNumber in range (0, len(tempImg)):
#         tempImg[pixelNumber] = np.uint8(round(tempImg[pixelNumber] * 255))

#     tempImg = tempImg.reshape(48, 48) # Reshape to be 2D

#     # load image from 2d array
#     img = Image.fromarray(tempImg, 'F').convert("RGB") # Must convert to be able to save

#     # imshow(np.asarray(img), cmap='gray')

#     img.save("resampled_dataset/train/" + str(y_resampled[i]) + "/" + str(i) + '.jpg', "JPEG",
#             quality=80, optimize=True, progressive=True)

# print("Done")

# # Step 4

# train_dir = './resampled_dataset/train/'

# train_datagen = ImageDataGenerator(rescale=1./255)

# print("Training:")
# # Sampled training set
# training_set = train_datagen.flow_from_directory(train_dir,
#                                                  batch_size=batch_size,
#                                                  target_size=(48,48),
#                                                  shuffle=True,
#                                                  color_mode='grayscale',
#                                                  class_mode='categorical')

```

Define a CNN model and compile it

```
model = Sequential([
    # Convolutions and pooling
    Conv2D(32, (3,3), activation='relu', input_shape=(48,48,1), activity_regularizer=l2(0.0001)),
    Dropout(0.15),
    MaxPooling2D((2, 2)),

    Conv2D(128, (2, 2), activation='relu', activity_regularizer=l2(0.0001)),
    Conv2D(128, (2, 2), activation='relu', activity_regularizer=l2(0.0001)),
    Dropout(0.15),
    MaxPooling2D((2, 2)),

    Conv2D(256, (3,3), activation='relu', activity_regularizer=l2(0.0001)),
    Dropout(0.15),
    MaxPooling2D((2, 2)),

    # Flatten to 1D data
    Flatten(),

    # Dense
    Dense(84, activation='relu', activity_regularizer=l2(0.0001)),
    Dropout(0.15),

    # Output
    Dense(num_classes, activation='softmax', activity_regularizer=l2(0.0001))
])

#####

# BASE LeNet-5
# model = Sequential([
#     # Convolutions
#     Conv2D(filters=6, kernel_size=(5, 5), activation="tanh", input_shape=(48,48,1), strides=1),
#     AveragePooling2D(pool_size=(2, 2), strides=1),

#     Conv2D(filters=16, kernel_size=(5, 5), activation="tanh", strides=1),
#     AveragePooling2D(pool_size=(2, 2), strides=1),

#     Conv2D(filters=120, kernel_size=(5, 5), activation="tanh", strides=1),

#     # Flatten to 1 dimensional
#     Flatten(),

#     # Fully connected layers
#     Dense(units=84, activation="tanh"),
#     Dense(units=6, activation="softmax"),
# ])

model.compile(optimizer=SGD(lr=0.01),
              loss='categorical_crossentropy',
              metrics=["accuracy"])

model.summary()
```

Define a function for log folders for TensorBoard (not used for this assignment but left in as it's useful)

```
# Create a function that creates a unique folder for training logs
root_logdir = os.path.join(os.getcwd(), "assignment_2_logs")
def get_run_logdir():
    run_id=time.strftime("run_%Y_%m_%d_%H_%M_%S")
    return os.path.join(root_logdir,run_id)

run_logdir=get_run_logdir()
print(run_logdir)
```

Define callbacks and fit the model

```
seed(1) # Helps a bit with randomness of the model

# Create checkpoint callback that saves every epoch
checkpoint_cb = keras.callbacks.ModelCheckpoint("final_model.h5",
    save_best_only=True, verbose=1, mode='min', monitor='val_loss')

# Create early stopping callback
early_stopping_cb = keras.callbacks.EarlyStopping(monitor='val_loss',
    min_delta=0, patience=6, verbose=1,
    restore_best_weights=True)

# Reduce learning rate on plateau callback
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.3,
    patience=4, min_lr=0.0001, verbose=1)

# Make a new log for the time of this run and use it for a tensorboard callback (not used, left
in as it's useful)
tensorboard_cb=keras.callbacks.TensorBoard(get_run_logdir())

# Use sklearn to get class weights as a list
# class_weights_list = class_weight.compute_class_weight(
#     'balanced',
#     np.unique(training_set.classes),
#     training_set.classes)

# Convert into a dictionary in the format of class1:weight1, class2:weight2...
# class_weights_dict = {}

# For each class, add it and its value to the dictionary
# for class_num in range(0, num_classes):
#     class_weights_dict[class_num] = class_weights_list[class_num]

steps_per_epoch = training_set.n // training_set.batch_size
validation_steps = validation_set.n // training_set.batch_size

hist = model.fit(x=training_set,
    validation_data=validation_set,
```

```

        epochs=500,
        callbacks=[tensorboard_cb,
                    checkpoint_cb, early_stopping_cb,
                    reduce_lr],
        steps_per_epoch=steps_per_epoch,
        validation_steps=validation_steps,
        #class_weight=class_weights_dict
    )

n_epochs = len(hist.history['loss'])

```

Evaluate the trained model

Firstly, plot the accuracy and loss of the model

```

mpl.rcParams['figure.dpi'] = 300

# Plot the accuracy and validation accuracy scores
plt.plot(hist.history['accuracy'])
plt.plot(hist.history['val_accuracy'])

# Plot early stopping line and text
plt.plot([n_epochs - 1, n_epochs - 1], [0.0, 1.0], 'k--', lw=1.5, c="#ff0000")
early_stopping_text = plt.annotate(f'Early stopping: epoch {n_epochs - 1}', xy=(0.91, 0.035),
                                   xycoords='axes fraction', rotation=90, c="#ff0000", fontsize=8)
early_stopping_text.set_bbox(dict(facecolor='ffffff', alpha=0.75, edgecolor='ffffff'))

plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Accuracy', 'Validation Accuracy'], loc='upper left')

plt.xlim([0, n_epochs]) # Show up to the number of epochs
plt.ylim([0, 1]) # 0.0 to 1.0 accuracy

plt.grid()
plt.rc('grid', color='grey', lw=0.1)

# Force integer epoch numbers and multiples of 10
ax=plt.gca()
f = lambda x,pos: str(x).rstrip('0').rstrip('.')
ax.xaxis.set_major_locator(mpl.ticker.MultipleLocator(10))
ax.xaxis.set_major_formatter(mpl.ticker.FuncFormatter(f))

plt.show()

mpl.rcParams['figure.dpi'] = 300

# Plot the loss and validation loss scores
plt.plot(hist.history['loss'])

```

```

plt.plot(hist.history['val_loss'])

# Plot early stopping line and text
plt.plot([n_epochs - 1, n_epochs - 1], [0.0, 10.0], 'k--', lw=1.5, c="#ff0000")
early_stopping_text = plt.annotate(f'Early stopping: epoch {n_epochs - 1}', xy=(0.91, 0.035),
                                   xycoords='axes fraction', rotation=90, c="#ff0000", fontsize=8)

early_stopping_text.set_bbox(dict(facecolor='ffffff', alpha=0.75, edgecolor='ffffff'))

plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Loss', 'Validation Loss'], loc='upper left')

plt.xlim([0, n_epochs]) # Show up to the number of epochs
plt.ylim([0, 2]) # 0.0 to 1.0 loss

plt.grid()
plt.rc('grid', color='grey', lw=0.1)

# Force integer epoch numbers and multiples of 10
ax=plt.gca()
f = lambda x, pos: str(x).rstrip('0').rstrip('.')
ax.xaxis.set_major_locator(mpl.ticker.MultipleLocator(10))
ax.xaxis.set_major_formatter(mpl.ticker.FuncFormatter(f))

plt.show()

```

Produce a classification report for the model's performance

This first requires getting the data in the right format for the classification report. List of predicted and true class labels (y) is created. `test_set.__getitem__(i)[1]` is used to get class labels (data is [0], class is [1])

```

# Predicting the classes of images

# Load the model
trained_model = keras.models.load_model("final_model.h5", compile = True)

# Invert the dict generated by imagedatagenerator so that the number is the key and the name is
the value
class_num_and_names = {v: k for k, v in test_set.class_indices.items()}

# Get the actual classes
true_classes = []

# Get the predicted classes
# Using this to get each item one by one and use argmax to get an integer value from one-hot en
coded result
# The normal .classes or .labels return *sorted* results, which are not what we want
for i in range(test_set.__len__()):
    true_classes.extend(
        np.argmax(test_set.__getitem__(i)[1], axis=1)
    )

```

```

# Get the predicted classes
predictions = trained_model.predict(test_set)
pred_classes = list(np.argmax(predictions, axis=1))

# Map the integer values to class names
pred_classes = list(map(class_num_and_names.get, pred_classes))
true_classes = list(map(class_num_and_names.get, true_classes))

# Put into a dataframe
actual_vs_pred = pd.DataFrame({'Actual': true_classes, 'Predicted': pred_classes})

mpl.rcParams['figure.dpi'] = 200

# Display a graphical confusion matrix, followed by a classification report
confusionMatrix = pd.crosstab(actual_vs_pred['Actual'], actual_vs_pred['Predicted'],
                              rownames=['Actual'], colnames=['Predicted'])

sn.set(font_scale = 0.8) # Label size
sn.heatmap(confusionMatrix, annot=True, annot_kws={"size": 8}, fmt="d") # Font size for numbers
plt.show()

print(classification_report(actual_vs_pred['Actual'], actual_vs_pred['Predicted'], zero_division="warn"))

```

Produce a receiver operating characteristic curve for each class and calculate area under the curve scores

As with the classification report, this required getting the data in the right format

```

sn.set(style='white', font_scale=1)

n_classes = len(class_num_and_names.keys())

# Blank lists ready to be filled
roc_true = []
roc_pred = []

# Get the predicted classes
# As above in the previous cell, except without the argmax
for i in range(test_set.__len__()):
    roc_true.extend(
        test_set.__getitem__(i)[1] # [1] is the class label
    )

# Get the predicted classes from what was worked out in the previous code cell
roc_true = np.array(roc_true)
roc_pred = np.array(predictions) # Use the predictions from the previous cell

```

```

# For each face, which is a list of probabilities of being each class, perform argmax to find the most likely.
# From this, replace everything with 0 except the most likely with 1
for i in range(0, len(roc_pred)):
    predicted_index = np.argmax(roc_pred[i], axis=0)
    roc_pred[i] = [0] * num_classes # Make a list full of 0s of length num_classes
    roc_pred[i][predicted_index] = 1 # Set the class with the highest probability to 1

# Colours to be used for each line
class_colours = ["#c20000", "#c2c200", "#00c200", "#00c2c2", "#0000c2", "#c200c2", "#999999"]

# Compute ROC curve and ROC area for each class

fpr = dict()
tpr = dict()
roc_auc = dict()

# ROC score for each class
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(roc_true[:, i], roc_pred[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area
fpr["micro"], tpr["micro"], _ = roc_curve(roc_true.ravel(), roc_pred.ravel())
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# Plot ROC curve for micro average
plt.figure()
plt.plot(fpr["micro"], tpr["micro"], c="#000",
         label='Micro-average ROC (AUC = {0:0.2f})'
         ''.format(roc_auc["micro"]))

# Plot ROC curve per class
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], c=class_colours[i], label='ROC curve of class {0} (AUC = {1:0.2f})'
             ''.format(class_num_and_names.get(i, "Error"), roc_auc[i]),
             linewidth=0.85)

plt.plot([0, 1], [0, 1], 'k--', label='50%', linewidth=0.85) # The dotted diagonal line

# Set limits of axes = make the height go slightly past 1 to account for the line thickness
plt.xlim([-0.0025, 1.0])
plt.ylim([0.0, 1.005])

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Facial Expression: Emotion Classification')
plt.legend(loc="lower right", prop={'size': 7})
plt.show()

```

Finally, visualise the model

```

tf.keras.utils.plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer_names=False, dpi=100)

```

Appendix B

The data resampling process was as follows:

1. Load the dataset with ImageDataGenerator into test, train, and validation sets.
2. From the train set, separate each image object into its data (X) and class label (y). Store all image data in one list, and all class labels in another.
3. Define and fit the resampler as shown in Figure B-1.
4. Save the resampled data back into a separate dataset directory as JPG images.
5. Read this newly saved data using ImageDataGenerator into a new train set.

The above steps were taken as the ImageDataGenerator's data was not compatible with Imbalanced-Learn library functions.

```
sampler = SMOTE(sampling_strategy='not majority') # Resample all except the
# sampler = RandomOverSampler(sampling_strategy='not majority') # Resample
# sampler = RandomUnderSampler(sampling_strategy='not minority') # Resample
X_sampled, y_sampled = sampler.fit_resample(train_images, train_classes)
```

Figure B-1: Image sampling code

AustinH ML2 FINAL Report

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

90/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7



Comment 1

A very good review of the dataset and its issues.

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17



Comment 2

nice range selected. Good

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

GRADING FORM: SMU GENERIC_GJ

AUSTIN HOOPER**89.50****RESEARCH-INFORMED LITERATURE**

Excellent effort again Austin. Excellent referencing. Relevant topics identified and applied. Good research to support design and evaluation and well communicated and applied.

9**KNOWLEDGE AND UNDERSTANDING**

A thorough, sound understanding demonstrated and applied. Key ideas discussed and uptodate research used here.

13.5**ANALYSIS**

A most thorough analysis and your discussions identify weaknesses in some methods. Excellent. Dataset bias analysis excellent too. You're not afraid to remove poorly represented emotions (as suggested in the assignment).

26.5**PRACTICAL APPLICATION AND DEP.**



Again, a detailed, thought out approach using relevant Python libraries and methods to process datasets, train and evaluate. Well done.

36.5

SKILLS FOR PROFESSIONAL PRAC.



An excellent report, well structured with helpful diagrams. REfs used appropriately and are of a high quality.

4